An Artificial Intelligence for the Board Game ‘Quarto!’ in Java

Jochen Mohrmann[1,2], Michael Neumann[1,2], David Suendermann[2]
[1]Hewlett-Packard, Böblingen, Germany
[2]Baden-Wuerttemberg Cooperative State University (DHBW), Stuttgart, Germany
{jochen.mohrmann,michael.neumann}@hp.com,david@suendermann.com

Abstract

This paper presents an artificial intelligence (AI) for the board game ‘Quarto!’ in Java. The program uses depth-first search for decision making. To improve runtime performance, we used alpha-beta pruning, a transposition table, and a Java constraint solver. The result of our work is an open-source Java program capable of beating human opponents in real-time.

Categories and Subject Descriptors I.2.8 [*Artificial Intelligence*]: Problem Solving, Control Methods, and Search

General Terms Artificial Intelligence, Alpha-Beta Search

Keywords NegaMax, Alpha-Beta Search, Choco, Constraint Programming in Java, Quarto

1. Introduction

‘Quarto!’ is a two-player board game which is a somewhat sophisticated version of the pen-and-paper game tic-tac-toe. There are 16 wooden pieces that can be placed on a board with four times four fields. Each of the pieces has four binary properties: large or small, white or black, round or angular, and hollow or solid. Figure 1 shows the board with all pieces. Both players alternately choose a piece for the opponent to place on an arbitrary field of the board. I.e., a turn consists of placing a given piece on the board and then choosing one for the opponent. The first player to complete a horizontal, vertical, or diagonal sequence of four pieces identical in at least one property, wins.

Even though the ‘Quarto!’ player community is rather small, the game has some interesting characteristics when it comes to developing an artificial intelligence (AI) using Java. To the best of our knowledge, there are no publications in relevant literature dealing with building an AI for the game ‘Quarto!’ up to now.

The present paper is structured as follows: In Section 2, we compare ‘Quarto!’s properties with other two-player games like chess. Sections 3 and 4 discuss details on the implementation of the AI in Java and how we optimized its runtime behavior. Finally, Section 5 will analyze the program’s performance w.r.t. computing time and strength of the AI against human players before concluding and outlining our future endeavors in Section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPPJ’13, September 11–13, 2013, Stuttgart, Germany.
Copyright © 2013 ACM 978-1-4503-2111-2/13/09…$15.00.
http://dx.doi.org/10.1145/2500828.2500842

Figure 1. Quarto board with all 16 pieces

2. Classification of the game

In game theory, ‘Quarto!’ is classified as zero-sum game with perfect information for two players [5]. The term *zero-sum game* refers to the outcomes of all players summing up to zero, i.e. a win for Player A automatically means a loss for Player B. Therefore, in the following, the outcome of ‘Quarto!’ will be denoted as ’-1’, ’0’ or ’1’ meaning loss, draw or win. The term *perfect information* refers to the fact that all players know the state of the game and all possible actions at any point in time. What makes ‘Quarto!’ special, is that players do not have their own pieces but share all available pieces.

The ‘Quarto!’ AI we developed can be classified according to [6] as a utility-based agent. It uses a heuristic function to evaluate the outcome of all possible states in the game up to a certain depth level of the game tree. From the classification of the game, the PEAS (Performance, Environment, Actuators and Sensors) Classification for the AI can be derived. The performance measure is the outcome of the game, which can be a loss, draw or a win. The environment is made up of the board and pieces. It is fully observable, deterministic, episodic, static and discrete. If two AIs play the game against each other, the environment can be interpreted as a competitive multiagent system. The actuators are the decisions where to place a piece and which piece to choose. At last, the sensors are logical representations of the four times four grid along with the position of all sixteen pieces.

3. Implementation of the 'Quarto!' AI in Java

In this section, we describe the two main components of the 'Quarto!' AI we developed. First, the tree search algorithm we used (alpha-beta pruning) is presented, and the second part deals with the implementation of a heuristic function required for the used tree search.

3.1 NegaMax Alpha-Beta Pruning

By regarding all possible moves that can follow a given game state as the latter's children, one can interprete the set of all possible states of 'Quarto!' as a search tree, or, more generally, graph. In this work, we considered three graph search methods:

a) The first approach we analyzed is to simulate all possible games once and store them in a database along with their expected reward. Then, at runtime, the AI has to look up the current state and the associated outcomes for all possible successor moves to decide what action to take. The problem of this implementation is the large number of possible states. Since there are 16 pieces that can be placed on 16 fields on the board, the number of different states is $16!^2 \approx 4.4 \cdot 10^{26}$ (not taking symmetries into account). Even when considering possible reductions of the search space (cf. Section 4), the simulation of all possible games is not feasible without high-performance computers.

b) A dynamic strategy we considered is a variant of the minimax algorithm called alpha-beta search. In order to make a decision, the complete tree is searched from the current state to find the path with the best outcome. In doing so, alpha-beta search prunes branches which certainly would not be reached. This can reduce the effective branching factor b to $\sqrt{b}$ in the best case [6]. For 'Quarto!' having an average branching factor of $b_a = 8.5$, this would result in a reduction by up to 66%. The average branching factor b_a can be calculated as follows, whereby p is the number of plies in the search tree and b_p the branching factor at ply p.

$$b_a = \frac{1}{p} \sum_{i=1}^{p} b_p \tag{1}$$

Equation 1 takes not into account that a number of games end before reaching the deepest ply. With the best reduction possible, the resulting number of states in the search tree can be approximated with equation 2.

$$\sqrt{b_a}^{p} = \sqrt{8.5}^{32} = 7.4 \cdot 10^{14} \tag{2}$$

However, this rate can only be achieved if the search algorithm examines the possible moves in a certain order. Since it is hard to tell which move should be examined next, the effective branching factor reduction is less.

c) The third strategy is to search the game tree up to a certain depth level and make a decision based on a heuristic evaluation function [5]. This approach is popular in the creation of AIs for multiplayer games (such as chess [7]) as it allows to control the complexity of the search tree.

Alpha-beta search is based on the *minimax* principle that takes advantage of the players' goal to *max*imize their reward thereby *min*imizing the opponents' (zero-sum). Minimax assumes the opponent to play perfectly within the range of given information and applies a depth-first search where it maximizes the accumulated reward at players' turns and minimizes it at the opponents'. Alpha-beta search comes up with the same answers, but usually in shorter time. Assuming that the opponent will not choose an action leading to lower reward than the guaranteed reward of another path, the evaluation of the path and the whole sub-tree can be skipped, without losing information [6].

Alpha-beta search uses a window that is set by the values alpha and beta. While alpha is the minimum value guaranteed, beta is the maximum accessible value that the opponent has to grant. In the beginning, these values build up a range from minus to plus infinity, but they are subject to update. As soon as a state is evaluated, its reward is compared to the boundaries of the search window. If the calculated reward of an action exceeds the beta value while maximizing, the calculation of the sub-tree can be omitted since the opponent is able to prevent this situation. If a reward undercuts the alpha value while minimizing, the sub-tree is omitted for the analog reason.

As 'Quarto!' is a zero-sum two-player game (cf. Section 2) the negamax variant of the alpha-beta search can be used to evaluate a game state regardless of its owner, i.e. the player who controls the board and wants to maximize the reward. A state is defined as the board with sixteen positions and a number of pieces placed on certain positions. After choosing the piece to be placed next, additionally, this piece is also included in the state, see Figure 2. While

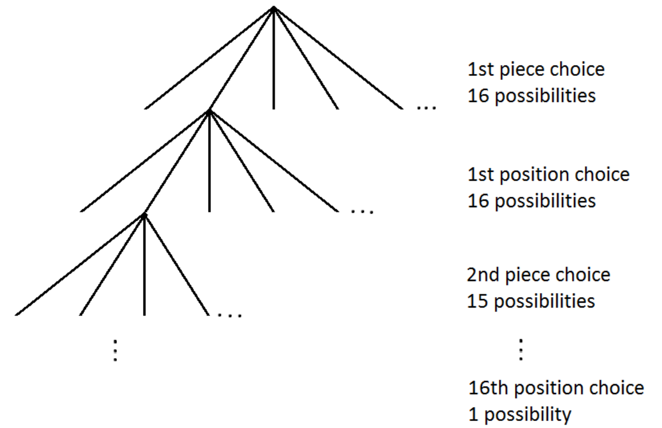

Figure 2. Game tree of 'Quarto!'

the conventional alpha-beta search implementation swaps between maximization and minimization of the rewards, the negamax variant always maximizes the outcome but alternately inverts the search window and negates results. In 'Quarto!', a player takes two actions during one round, namely placing a piece on the board and choosing the next one. Only after completing both actions, negation is carried out. After applying negamax alpha-beta search to the current state, the winning path is determined. If a transposition table is used (cf. Section 4.3), a table lookup is carried out to determine whether an alpha-beta search had been executed on the current state before in which event the winning path is already known.

One of our goals is to implement an AI that responds within an acceptable time frame. For a progressed game, the negamax algorithm may find the perfect solution within time, but for a nearly empty board, the number of moves to make and the possible options to take is too high, even with optimizations. Therefore, the algorithm needs to stop searching at a certain depth level and uses a heuristic evaluation function for states that are neither wins, losses, nor draws.

3.2 Implementation of a heuristic evaluation function

Obviously, in 'Quarto!', the higher the number of lines that can be completed with a given piece (rows, columns, or diagonals), the higher the chance that one of these lines can be ultimately

completed. So, depending on whose turn is to be evaluated, the number of such possible lines or its negation is expected to serve well as a heuristic function. By comparing the heuristic values to the value of a draw (0), the AI may tend to enforce a draw or take a risk in a certain direction.

In particular, our AI uses the number of lines with three pieces of an identical property as heuristic value. Thus, the value is in the range of zero to seven. A game with no combinations of three pieces of identical properties in a line has the same value as a draw (0).

4. Optimizations

The game 'Quarto!' has some properties which can be used to substantially reduce computational complexity. We considered a number of symmetries inherent to the game and also looked at transposition tables to memorize states evaluated before.

4.1 Finding Symmetries with the Java constraint solver Choco

We considered three kinds of symmetries in 'Quarto!': field symmetries and two kinds of piece symmetries (characteristic and binary). By resolving these, different states can be mapped to indistinguishable equivalence states whose number is much smaller than that of all possible states. Equivalent boards are determined by rotating or mirroring. Altogether, each equivalence board subsumes 32 symmetric boards. In order to determine the set of all board symmetries, we used constraint programming [4] as discussed in the following.

One way to find all board symmetries is to rearrange the pieces on the fully occupied board in all possible ways and save lineups that share the same structure as the original lineup. This structure is composed of ten combinations (four possible full lines along rows, four along columns, and two along diagonals). Two game boards are equivalent if each combination of four pieces on the original board is present on the equivalence board. Hereby, the order within the four pieces does not matter.

Assume $F = \{0, \ldots, 15\}$ to be the set of the 16 available pieces, one could uniquely represent an arbitrary subset of pieces $F_{\text{sub}} \subseteq F$ by means of the sum formula

$$\varphi_{\text{sub}} = \sum_{f \in F_{\text{sub}}} 2^f. \tag{3}$$

An example of an arrangement of 16 pieces on the board together with the φ coefficients of the ten full lines is shown in Table 1.

The objective of our search for board symmetries is to determine all possible arrangements of pieces, such that the set of ten φ coefficients are the same as on the original board, but not necessarily in the same order. Table 2 shows an arrangement resulting in the same set of φ coefficients like Table 1 by way of flipping multiple rows and columns.

				4680
2^0	2^1	2^2	2^3	15
2^4	2^5	2^6	2^7	240
2^8	2^9	2^{10}	2^{11}	3840
2^{12}	2^{13}	2^{14}	2^{15}	61440
4369	8738	17476	34952	33825

Table 1. Full 'Quarto!' board with φ coefficients

The computational cost of an iterative approach examining $16! \approx 2 \cdot 10^{13}$ valid piece arrangements is quite high. Considering the problem as a Constraint Satisfaction Problem (CSP) [6] can

				33825
2^9	2^8	2^{11}	2^{10}	3840
2^{13}	2^{12}	2^{15}	2^{14}	61440
2^1	2^0	2^3	2^2	15
2^5	2^4	2^7	2^6	240
8738	4369	34952	17476	4680

Table 2. Alternative full 'Quarto!' board with the same φ coefficients

save us a lot of time. Constraint programming is a programming paradigm where a problem is described in a declarative way using predicate logic [4]. Constraints are logical formulas describing relations between variables. In constraint programming over finite domains, these variables are limited to predefined integer domains. Considering our problem, the sixteen positions of the 'Quarto!' board can be denoted as variables P_n where the variable domain from zero to fifteen corresponds to the piece being placed on the particular field. Since each piece is unique the first constraint is that all position variables P_n are mutually different. Now, we need another ten variables R_m denoting the sums of the rows, columns and diagonals calculated with Equation 3. The domain of these variables is composed of the ten values shown at the bottom row and the right column of Table 1. They also have to be mutually different because no combination of four pieces can appear more than once on the same board. The last constraint is that each value of R_m has to be the sum of the values in one row, column or diagonal. Each allocation of the position variables P_n that satisfies all constraints is a valid equivalent of the original board and represents one board symmetry.

For the implementation of a constraint solver in Java, we used the open source library Choco [1]. Since constraint programming is a paradigm coming from the realm of logic programming and seems to be used rather for teaching and research than for industry purposes, the capabilities of Java libraries like Choco, JaCoP [2] or firstcs [3] are somewhat limited compared to logic programming languages like Prolog. For example, with JaCoP it is hard to define non-continuous variable domains. On the other hand, Choco implements relatively few kinds of constraints. A major drawback we found, compared to Prolog, is the lack of applying arithmetic functions on constraint variables. With the libraries we used, it is not possible to do calculations like adding an integer to a constraint variable because they have different data types. But there are some benefits that make Choco a suitable choice for the problem of resolving all equivalent 'Quarto!' boards. The first one is that it is easier to work with a Java library instead of an interface to another programming language. The second benefit in our case is a huge performance gain provided by Choco compared to a similar solution we implemented in Prolog using the clpfd module (Constraint Logic Programming over Finite Domains). With Prolog, the 32 board symmetries were found in about 2 hours, whereas the Choco implementation on the same machine took only 30 seconds to complete.

Besides the described field symmetries there are two piece symmetries which we found in 'Quarto!'. We call them characteristic symmetry and binary symmetry. The first one presents the fact that the characteristics of the pieces are interchangeable (it does not matter whether there is a line with three pieces of equivalent color in a row or whether they are of equivalent size). As a result there are $4! = 24$ equivalent arrangements of the characteristics. The binary symmetry toggles the value of different characteristics for all pieces on the board (it does not matter whether there are three black

pieces in a row or three white ones). There are $2^4 = 16$ different possible combinations.

4.2 Using symmetries to reduce the search space of the game tree

In our implementation, the state of the game is represented by the current board together with the next piece to place. This is embodied by an integer array of length 17. Each of the first 16 elements is either occupied by a piece, represented by its index from zero to fifteen or empty which is encoded by the value sixteen. The 17th element presents the index of the next piece to be placed. The index of a piece is the decimal equivalence of the four binary characteristics of the piece from 0000_2 to 1111_2. The array can be viewed as a 17-digit number, each element representing one digit to the basis of seventeen. The most significant digit is the first element of the array, the least significant the piece to be placed. Now we want to determine one representative out of all symmetrical equivalences to this setting. In order to get always the same representative we look for the equivalent setting that minimizes the described number. This way, it is guaranteed that there is only one representative for each equivalence class, though it might be obtained using different transformations.

Each possible combination of the three symmetries described in Section 4.1 is applied to the original board. The value of a digit is calculated the following way. First, field symmetry is applied. The piece value of the corresponding element using a certain field symmetry is looked up in the original array. If it is not sixteen (empty field), piece symmetries are applied. According to the field symmetry, digits are exchanged and joined with the binary symmetry value using the logical XOR operation. Beginning with the most significant one, digits are analyzed one after another. If a digit of an equivalence exceeds the lowest transformation value that was found so far, the next combination of symmetries is evaluated. If, however, the value falls below, a better combination is found. If the values are identical, the next digit is evaluated. The transformed state is used for further evaluation of the board. The resulting value for the reward is the same for each board of the equivalence class. In order to find the best move, the value of the position or piece has to be transformed back to the original setting using reverse transformation functions.

4.3 Using a transposition table

Since the rules of 'Quarto!' specify neither the order of pieces to be chosen nor the order of fields to place a piece on, the same board position can originate from different move sequences, called *transpositions*. Thus, the game tree is not an actual tree, but, more generally, a graph because branches can merge [5]. We use this property to accelerate the alpha-beta search. When evaluating a state, it is stored together with the calculated heuristic and the best decision option in a transposition table. This way, a state can be looked up in the transposition table first and only needs to be evaluated if there is no entry in the table yet. To improve the chance of encountering state in the transposition table, we also use symmetry equivalences during lookup and storing.

Using transposition tables along with alpha-beta search may lead to results not in line with the minimax algorithm's outcome. If a state is evaluated coming from a specific path, the search window has a certain size. If the search window leads to a cutoff, the value of the exceeded border can be saved in the transposition table. If, however, the same position is reached from a different path and with a different search window, the cutoff is not guaranteed. The saved value may not be in line with the best solution. In fact, the state may be in the path of an optimal solution. To minimize the influence of this problem, the value of nodes where a cutoff occurs is not saved in the transposition table. However, this does not guarantee that the value of an ancestor is not influenced by a cutoff of one of his descendants.Therefore results may differ depending on the particular search window.

5. Performance analysis

In order to evaluate the 'Quarto!' AI, we analyzed its performance along two dimensions: intelligence and speed. To assess the former, we carried out a subjective experiment with ten human players competing against the AI. To evaluate speed, we measured the program's mean execution time per turn in several objective experiments.

In all experiments, we used a notebook with an Intel Core i5 2-core CPU (1.6 GHz) and 4 GB RAM.

5.1 Subjective evaluation of 'Quarto!''s intelligence

As a preparation, each of the ten subjects, all of which were computer science students and seven of them novices to 'Quarto!', got an explanation of the game and its rules. Then, we demonstrated a sample run of the program to make the person more familiar with the user interface which is seen in Figure 3. After these instructions, each subject played five games with different settings of the program (different search depths). In Table 3, the results of the experiment are shown. The counts of wins and losses are related to the program. Overall, the 'Quarto!' AI won 34 out of 50 games.

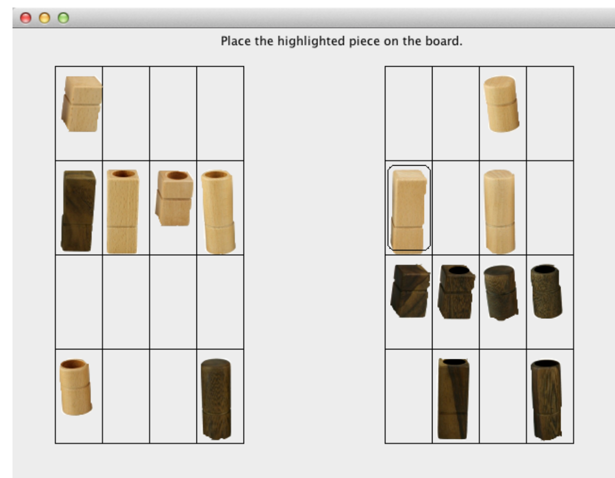

Figure 3. Screenshot of the graphical user interface

Search depth	Wins	Draws	Losses	Played games
2	5	0	8	13
5	16	0	5	21
6	13	0	3	16
$\sum$	34	0	16	50

Table 3. Overall winning statistics of the 'Quarto!' AI against human opponents

As shown in Table 3, the AI is more likely to lose a game with the search depth of two which is not surprising since this means that it looks ahead only one field and one piece choice. As expected, the winning rate of the AI increases with a growing search depth. For this experiment, we chose the values five and six because with these settings, the time of the moves can be still considered acceptable for playing as described in the following section.

We also analyzed how performance varied depending on whether players were experienced players or novices. It turned out that the three expert subjects had a higher winning rate with low search depths but the number of wins and losses with the search depth six was very similar to the persons without foreknowledge of the game. The detailed results are presented in table 4. This indicates that from a certain depth level of the alpha-beta search also experienced users have a hard time competing with the AI we build. In conclusion, this experiment states that we built an artificial intelligence for 'Quarto!' which is able to beat humans.

	Novices			Experts		
Search depth	Wins	Draws	Losses	Wins	Draws	Losses
2	5	0	5	0	0	3
5	14	0	1	2	0	4
6	8	0	2	5	0	1
$\sum$	27	0	8	7	0	8

Table 4. Winning statistics of the 'Quarto!' AI divided into categories of human expertise

5.2 Runtime performance evaluation

In Figure 4, the program's runtime performance is visualized as a function of the average time per turn over the search depth of the alpha-beta search. The underlying measured values are shown in table 5. As expected, execution time exponentially grows, indicated by an almost straight line on the logarithmic scale of the graph. This is in line with the number of possible board configurations growing exponentially when traversing more and more plies of the game tree. The time for a turn includes both placing a piece and choosing one for the opponent. Looking at the times for these two actions separately would create the impression that choosing a piece is much less time-consuming than placing one. This is caused by the lookup in the transposition table to search already seen board configurations which accelerates choosing a piece for the opponent as the knowledge about the tree search of the previous action of placing a piece was already prepopulated.

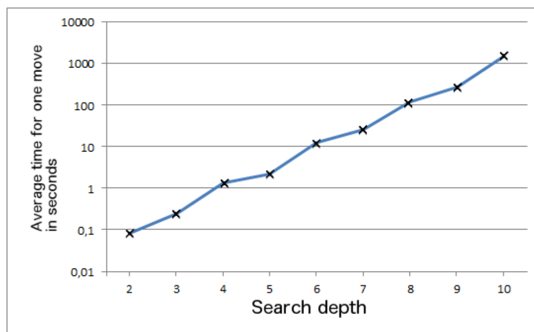

Figure 4. Runtime performance of the program (logarithmic scale)

A value of the search depth that produces a reasonable playing experience is five or maximal six. An average execution time of about 20 seconds per term may still be acceptable for the game since the human opponent needs a certain response time as well. A problem that can come up is that turns can significantly deviate from the average execution time depending on the game's history. During our experiment, the longest time needed for a turn with a search depth of six was 26.9 seconds as compared to the average time of a mere 12.5 seconds. Usually, execution time decreases towards the end of a game due to the lower number of possible alternatives of pieces and empty fields. Figure 5 presents the average execution time from turn to turn playing with a search depth of six. In 'Quarto!' the maximum number of one player's moves is eight. The graph in figure 5 depicts seven moves as this was the highest number of moves reached within our experiment.

Search depth	Time in seconds
2	0.1
3	0.2
4	1.4
5	2.2
6	12.5
7	25.3
8	118.6
9	272.8
10	1507.6

Table 5. Average execution time per move

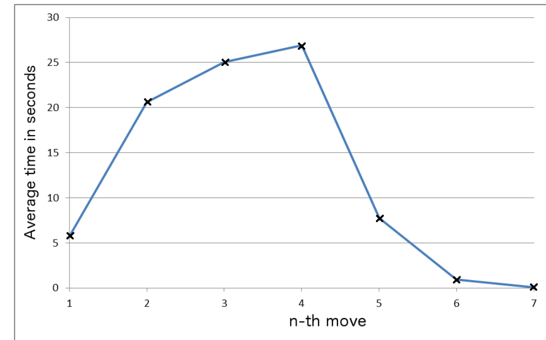

Figure 5. Execution time from turn to turn with search depth six

6. Conclusions and future work

The objective of this work was to develop an artificial intelligence for the board game 'Quarto!' in Java and to analyze it with respect to its computational behavior. As presented in the previous section, we were able to accomplished this goal. The resulting 'Quarto!' AI was able to beat the clear majority of human players. Among other things, we found that the application of constraint programming in Java can produce a substantial performance gain for the task at hand. For the sake of the interested reader, we decided to release the Java code of the presented 'Quarto!' AI as open source under the GNU General Public License (GPLv3). It can be downloaded from `http://sourceforge.net/projects/quartoagent`.

An outstanding issue is the interference of alpha-beta pruning with a transposition table described in Section 4.3. This problem might be solvable by additionally storing the search window of the alpha-beta search in the transposition table. A way to render the present AI even more effective is to further increase the search depth. The following optimizations could be added to this end: First, the order in which alternatives are evaluated could be aligned with information from the last search. Checking the best option from the last execution fist increases the likelihood of finding the best path directly and therefore increases the expected cutoffs. Another optimization would be to begin calculations during the opponent's turn. In addition, multithreading could be used to evaluate the board in a parallel manner and distributed across multiple

nodes. It has to be assessed how well these techniques combine with alpha-beta search.

Future work on the project in addition to the aforementioned improvements includes the development of a Java applet to allow for playing the game online. As a result, we could collect more statistical data from games against human players that could be used to tune the heuristic function and more thoroughly evaluate the AI's performance. Also, the release of a smartphone or tablet app based on the presented Java code is an option to achieve this goal.

Acknowledgments

We would like to thank all people who were involved in the development process and all participants of our experiment. Special thanks to Ingo Janz who was able to identify the last remaining piece in the puzzle of resolving symmetries with constraint programming.

References

[1] Choco. URL `http://www.emn.fr/z-info/choco-solver` [retrieved 03-24-2013].

[2] JaCoP. URL `http://jacop.osolpro.com` [retrieved 05-20-2013].

[3] M. Hoche, H. Müller, H. Schlenker, and A. Wolf. firstcs - A Pure Java Constraint Programming Engine. In M. Hanus, P. Hofstedt, A. Wolf, S. Abdennadher, T. Frühwirth, and A. Lallouet, editors, *Second International Workshop on Multiparadigm Constraint Programming Languages - MultiCPL03, 2003*.

[4] K. Marriott and P. Stuckey. Programming with Constraints: An Introduction. MIT Press, 1998.

[5] I. Millington and J. Funge. Artificial Intelligence for Games. CRC Press, 2nd edition, 2009.

[6] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 3rd edition, 2009.

[7] C. Shannon. Programming a Computer for Playing Chess. Philosophical Magazine, 7th Series, 41, No. 314, 1950.